# Coping With Problems Computers Can't Solve

J. Franco *
*Computer Science, ECECS*
*University of Cincinnati*
*Cincinnati, OH* 45221-0030
E-mail: franco@gauss.ececs.uc.edu

May 20, 1997



For over 40 years, the branch of computer science known as artificial intelligence has been unable to fulfill its promise of providing truly intelligent machines for general use. As early as the 1950s and 1960s, scientists developed computational models of intelligence then excitedly coded these models into the best computers of the day. At first the scientists were puzzled by the machines' inability to produce reasoned output. This turned to frustration when they realized they had smashed into a thick brick wall which they had failed to forsee. This wall stopped them in their tracks and continues to do so today, decades later. It also has stymied scientists and engineers in other fields such as operations research (the field concerned with determining efficient manufacturing and scheduling protocols), VLSI chip design and testing, and data base management to name a few. The brick wall exists because many combinatorial problems that are of fundamental importance to intelligent models, operations research, etc. are $\mathcal{NP}$-complete.

## 1 The Thick Brick Wall

Rather than define $\mathcal{NP}$-completeness, we illustrate the problem it causes with an example taken from manufacturing. Consider a general purpose welding device that is to be used on an assembly line to make numerous welds at pre-determined positions on a particular kind of part. The positions are welded in some sequence, called a *schedule*, which is programmed into the welder before a "run." It is conceivable that as many as 1000 welds may have to be made on each part that passes by the welder on the assembly line. Clearly, the speed of the line and therefore the cost of producing product depends on how fast the welder can finish all its welds on each part and return to its starting point. But the speed of the welder depends on the distances traversed from one position to the next and so on. What is needed is an optimal schedule for the positioning of the weld-head: that is, a schedule that minimizes the total distance traversed by the weld-head on each part.

*Supported in part by the Office of Naval Research, N00014-94-1-0382

Let's consider how we might find such an optimal schedule. A simple algorithm is the following: tabulate all possible schedules along with the total distance traversed and search for the schedule offering the least total distance. This seems like a perfectly reasonable approach at first, but when we try to implement it we run into the thick brick wall. The number of schedules that we have to tabulate can easily be seen to be the factorial of the number of positions needing welds (from here on we use $n!$ to denote this number where $n$ is the number of positions needing welds). If we need even as few as 30 welds, $n!$ is 30! which is greater than $10^{30}$. Suppose our implementation is run on a supercomputer that can tabulate schedules at the rate of one per $10^{-12}$ seconds (very, very fast!!). Then we can complete our task involving 30 welds in no less than $10^{30} * 10^{-12}$ seconds. But this is about 30 years. Clearly, this is unacceptable and we must find a better way.

## 2 The Thick Brick Wall Seems Impenetrable: Jackhammers Don't Work

Before we search for a better algorithm we must understand what is holding us back. The problem that we have with tabulating possible solutions is that, if $n$ is increased by 1, the size of the table, and therefore the amount of time needed to find an optimal schedule, more than doubles. This phenomenon is known as the *exponential explosion of complexity* (here, the word complexity refers to the running time of an algorithm). An algorithm suffering from this phenomenon has complexity at least $2^n$. What we need is an algorithm with complexity $n^2$ or, better yet, $n$. If we had a scheduling algorithm of complexity $n^2$ we could solve our scheduling problem for 1000 welds in less than 1 second with a Pentium processor. With a complexity of $n$ we could get a result in less than 1 second on an IBM XT.

How can we find an algorithm of complexity $n^2$ or $n$? We can achieve this complexity if there is a way to iteratively extend a partial solution in such a way that the extensions do not deviate from the optimal solution. Let's try the obvious algorithm, called the greedy method, for extending a weld schedule: start with a partial schedule of one weld position chosen arbitrarily; repeatedly append to the partial schedule the weld position that is closest to the last position in the partial schedule and that has not yet been added to the partial schedule; when all positions are in the partial schedule, return the partial schedule as a full schedule. The complexity of this algorithm is $n^2$ since the distance between all pairs of positions must be considered. The question is whether this algorithm always produces an optimal schedule.

We investigate this question by considering the simple example of Figure 1 which represents five weld positions denoted **A**, **B**, **C**, **D**, and **E**. Lines and numbers are used to show distances: a line designates a pair of positions and the number associated with that line gives the distance

between that pair[1].

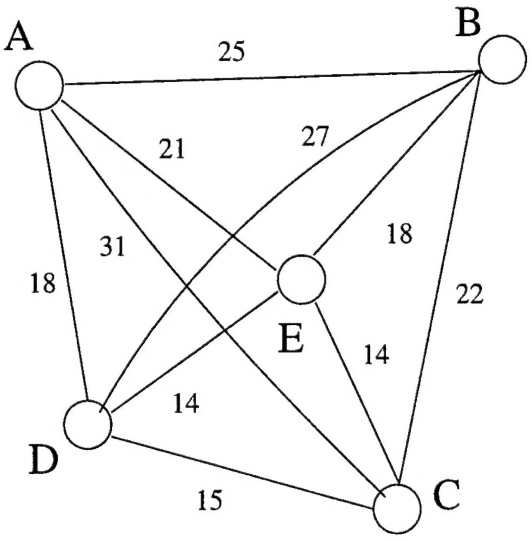



Figure 1: Finding an optimal weld schedule

We apply the greedy method to Figure 1, starting at point **A**, say. We create a partial schedule containing **A**, then append **D** to the schedule since the distance to **D** from **A** is 18 and this is less than the distance from **A** to any other unscheduled point. Next we append **E** because the lowest distance from **D** to an unscheduled point is 14 (to **E**). Next **C** and finally **B** are scheduled and the algorithm returns the schedule **A** - **D** - **E** - **C** - **B**. The total distance covered by this schedule is $18 + 14 + 14 + 22 + 25 = 93$. However, a better schedule would be **A** - **D** - **C** - **E** - **B** which covers a distance of $18 + 15 + 14 + 18 + 25 = 90$. This shows that our greedy method does not necessarily produce an optimal schedule.

Can some other $n^2$ complexity algorithm for obtaining optimal schedules be found? For this problem, no $n^2$, or $n^3$, or $n^{10}$ or even $n^{100}$ algorithm has been found despite decades of searching. In fact, it is generally believed that no $n^k$ algorithm, $k$ a constant, will ever be found for this problem (an algorithm of complexity $n^k$, where $k$ is a constant, is said to be *efficient*). This pessimism extends to all the $\mathcal{NP}$-complete problems, one of which is the weld scheduling problem, a variant of the well-known Traveling Salesman Problem. Unfortunately, most interesting real-world problems are $\mathcal{NP}$-complete. Therefore, this is a very serious situation and something has to be done about it.

## 3 Going Around the Thick Brick Wall: Approximations

Because so many important real-world problems are $\mathcal{NP}$-complete, it is pretty clear that something has to be done to cope with our inability to find efficient algorithms for these problems. The astute

[1]The curious reader will realize that the numbers given cannot represent euclidean distances if the five points are in the same plane. It is typically the case that the *time* needed to move an object is not proportional to euclidean distance because it depends on other factors such as direction of movement.

reader may be wondering whether parallel processors are the answer. Perhaps we can achieve real-time solutions to $\mathcal{NP}$-complete problems by dividing the computation required by an inefficient algorithm over countless numbers of processors, all of which are set to simultaneously grind away on their small piece of the computation and quickly report back a result.

Parallelization can help but cannot solve the problem altogether. The reason is there are not enough atoms in the universe. Assume that every processor requires at least one atom and that there are fewer than $2^{200}$ atoms in the universe[2]. A weld scheduling algorithm of complexity $2^n$ distributed evenly over $2^{200}$ processors (the maximum possible) operating at supercomputer speed would compute for more than one century if $n$ is only 235. Of course, to achieve this we would have to destroy the welder and parts just to get all the material needed to make all the processors that will be used for the calculation. Clearly, another approach is needed.

Another possibility is to make machines faster. Unfortunately, we run into a barrier here too. It turns out that due to Heisenberg's uncertainty principle it is unlikely that computers will ever reach speeds above a few GigaHertz. The reason is roughly the following: in order to get an output from a physical object, some physical quantity must be observed. But such observations perturb the object, making it hard to measure the quantity exactly. Thus, some time is needed to make sure of its value. This time prevents computers from going arbitrarily fast.

A strategy that often works is to use an algorithm of $n^2$ or $n$ complexity to find a solution that approximates the optimal solution. We saw, in discussing the example of Figure 1, that the greedy method returned a solution of total distance 93 but the optimal solution had value 90. Thus, the greedy method returned a solution that had value only 3% away from the optimal value. If we can guarantee that the greedy method can always find a solution of value that is less than 3% away from the value of the optimal solution, then the greedy method for finding optimal schedules might have tremendous practical use. Unfortunately, the best we can guarantee for optimal schedules, assuming distances are euclidean, is to be within 50% of the optimal value; this is, by the way, accomplished by an algorithm other than the greedy method. In the general case, when distances are not necessarily euclidean, efficiently finding an approximation that is guaranteed to be within a fixed percentage of the optimal is impossible unless there exists an efficient algorithm for finding the optimal solution.

Fortunately, good approximation algorithms exist for many $\mathcal{NP}$-complete problems. For example, consider the Bin Packing problem. The telephone company needs to store telephones of various sizes in a warehouse. The warehouse contains a number of bins of identical capacity. The question is how to organize the telephones in bins so as to use the minimum number of bins. A good efficient approximation algorithm for this problem is known as the *First-Fit Decreasing* (FFD) heurtistic. The idea is this: number bins $1, 2, 3, \ldots$ arbitrarily; repeatedly put the largest size, unstored tele-

[2]This is a reasonable guess

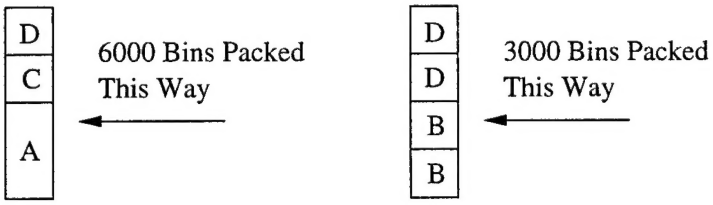
phone into the lowest number bin with remaining capacity greater than the size of the telephone (that is, the first bin in which the telephone will fit). It can be shown that FFD will never cause more than 22% extra bins to be used. Figure 2 gives an example showing that we can be off by as much as 22% when the number of telephones to store is large.

Input:

| Type | Size | Quantity |
|---|---|---|
| A | 501 | 6000 |
| B | 252 | 6000 |
| C | 251 | 6000 |
| D | 248 | 12000 |

**Bin Capacity:** 1000

Optimal Solution: 9000 Bins Needed




Approximation Using FFD: 11000 Bins Used


Figure 2: A Bin Packing example showing optimal solution and FFD approximation

Before continuing, we answer a question that frequently pops up at this time: how can you prove an approximation algorithm guarantees a certain performance if you don't know what the optimal solution is? Or, asking the question another way: if you can prove a performance guarantee, can't you then find the optimal solution? The answer to this question is no. We illustrate using the *First-Fit* (FF) heuristic which is FFD but without forcing consideration of largest size telephones first For FF, no more than one bin can be more than half empty since the telephones in the second such bin would have been placed by FF into the first such bin, at least. On the other hand, an optimal solution can do no better than completely fill all the bins needed for a solution. Therefore, FF produces a solution requiring no more than twice the number of bins taken by the optimal solution plus 1. It is left to the reader to supply examples showing that FF can really be this bad. Not all performance arguments are this simple; some require more than 50 pages. But this

argument shows how it is possible to compute performance guarantees without knowing the optimal solution.

# 4 Probabilistic and Empirical Results

Sometimes, the performance guarantees of approximation algorithms are much worse than is actually encountered in practice. Often, this is because locally optimal components can be constructed and these, when put together, can produce optimal or near-optimal solutions. The Bin Packing problem provides a great example of this. Suppose telephone sizes are uniformly distributed from size 0 to the bin capacity. That means any one size is as likely to exist as any other from size 0 up to the bin capacity [3]. Then, for any size greater than the mean, there is, with high probability, a size below the mean such that the sum of the two sizes is either exactly the bin capacity or extremely close. Each such pairing can fill a bin nearly or exactly to capacity. Probabilistically, we can find such pairings filling nearly all bins to capacity or just below capacity. It can be proven that, under a uniform distribution, only a constant number of bins are wasted, in probability, using FFD. The same results hold for a very wide variety of distributions and Bin Packing is regarded to be an easily solved problem.

However, many $\mathcal{NP}$-complete problems do not present an obvious way to combine locally optimal components, and algorithms for these may show good performance sometimes and poor performance other times. To illustrate, we return to the field of artificial intelligence and consider the Satisfiability (SAT) problem. An instance of SAT is a Boolean formula in Conjunctive Normal Form (CNF): that is, an "AND" of expressions which are "OR"s of Boolean variables and their complements. Figure 3 gives a sample CNF Formula.

$$(x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_2 \vee \bar{x}_3 \vee \bar{x}_6) \wedge (\bar{x}_3 \vee x_5 \vee x_6) \wedge (x_3 \vee x_4 \vee x_6) \wedge (\bar{x}_4 \vee \bar{x}_5 \vee \bar{x}_6)$$

**Figure 3: A Boolean Formula in Conjunctive Normal Form**

In this example, the Boolean variables are $x_1$, $x_2$, $x_3$, $x_4$, $x_5$, and $x_6$. Each can be assigned only one of the two values *true* and *false*. If variable $x$ is assigned value *true* (*false*) then its complement $\bar{x}$ has value *false* (*true*). For any variable $x$, both $x$ and $\bar{x}$ are called *literals* where (and if) they appear in a formula. Expressions such as $(x_1 \vee \bar{x}_2 \vee x_3)$ are called clauses. A clause has value *true* if and only if one of its literals has value *true*. A formula has value *true*, or is satisfiable, if and only if all its clauses have value *true* for some assignment of values to its variables. The SAT problem is to determine whether or not a given formula is satisfiable. The assignment $x_1 = x_2 = x_4 = x_5 = x_6 = true$, and

[3] This, of course, is not reasonable but the ideas we use here can be generalized to support similar results under more reasonable distributions.

$x_3 = false$ causes all clauses of the formula in Figure 3 to have value *true*. On the other hand, any assignment with $x_1 = x_3 = false$, and $x_2 = true$ causes the first clause to have value *false*.

SAT is probably the most well-known $\mathcal{NP}$-complete problem of all. It was the first problem shown to be $\mathcal{NP}$-complete and has been used to show others are $\mathcal{NP}$-complete. In real-life it shows up in VLSI testing and design, in theorem proving, in preventing catastrophic machine shutdowns in case of part failure plus many other aspects of science, engineering, operations research, and artificial intelligence.

There is no known way to combine locally optimum components to solve a general SAT problem; there seems to be a deep interdependence between large subsets of clauses, especially when formulas are not satisfiable. SAT is further handicapped by the fact that the concept of approximation algorithm is meaningless: either an input formula is satisfiable or not and we must determine precisely which or we have made a big mistake. Therefore, SAT is not regarded as an easy problem, in general, and current research aims to improve this situation.

One research direction attempts to uncover special classes of SAT that can provably be solved by an efficient algorithm. Remarkably, there are many such classes. Here we mention a few.

If a formula is restricted so that every clause contains at most two literals, it can be solved in time proportional to the size of the formula by a special algorithm. Such formulas are called 2-SAT formulas. It is interesting to note that formulas containing at most three literals per clause (3-SAT) are $\mathcal{NP}$-complete. Noone really understands why adding one more literal to each 2-SAT clause has such an effect on complexity: adding a literal to clauses containing one literal changes nothing in this regard.

If a formula is restricted so that at most one literal in every clause is not negated, then the formula can be solved in time proportional to the size of the formula. Such formulas are called *Horn* formulas. Several efficiently solved classes are extensions of Horn formulas. There are also hierarchies of formulas such that a formula at level $k$ in the hierarchy can be solved in time $2^k$ times the square of the length of the formula. If there is an efficient way to check whether a formula belongs to such a special class of SAT, this check can be applied before using a general purpose SAT solver. In case the check succeeds, a special purpose and efficient algorithm can be brought to bear.

It seems that identifying efficiently solved special classes of SAT helps some but not much. In particular, this approach usually fails when input formulas are not satisfiable. This is illustrated by probabilistic and empirical results obtained for the distribution of formulas of $n$ variables, $m$ clauses, and three literals per clause where each clause is equally likely to be any of the $8\binom{n}{3}$ different possibilities obtained from $n$ variables. Thus, this model generates random 3-SAT formulas. Experiments have shown a random formula is satisfiable, in probability, if the parameters of the distribution are set so that $m/n < 4.2$, and is not satisfiable, in probability, if parameters are set

so $m/n > 4.3$. We know, through probabilistic analysis, that there is a searching algorithm that finds a satisfying assignment efficiently with high probability when $m/n < 3.003$. But, so far, a random formula is known to be a member of an efficiently solved special class of SAT, with high probability, only if $m/n < 1$.

There is no known algorithm that efficiently verifies the unsatisfiability of a random formula with high probability when $m/n$ is a constant greater than 4.3 or even when $m/n$ grows as fast as $n^\epsilon$, $\epsilon < 1$. This seems to be a consequence of the following statement which can be proven by induction: a minimally unsatisfiable set of $p$ clauses[4] must contain fewer than $p$ distinct variables (negated or unnegated occurrences are treated the same). A simple, efficient algorithm verifies unsatisfiability when $m/n > n$, in probability, but random formulas generated when parameters are set this way are not, with high probability, a member of a known, efficiently solved special class of SAT.

No one has been able to find a good algorithm (in some probabilistic sense) for verifying unsatisfiability for a vast range of formula types. Even *resolution* fails to do a good job. One of the achievements of probabilistic analysis is that it shows exactly why resolution fails: this has to do with the "sparsity" of random 3-SAT formulas. However, probabilistic analysis has yet to help find a method that succeeds. Research on this question is ongoing and involves people from all over the world. If a fast method, in some probabilistic sense, is found, the result might have great impact on the field of artificial intelligence as well as other fields. But it is likely, given all the thought that has gone into this question, that a successful algorithm will be unlike any other ever thought of.

## 5 Read More About It

1. A very good discussion of the theory of $\mathcal{NP}$-completeness and approximation algorithms is Michael R. Garey and David S. Johnson, *Computers and Intractability: A Guide to the Theory of $\mathcal{NP}$-completeness*, W. H. Freeman, San Francisco (1979), ISBN: 0-7167-1045-5.

2. For a description of Heisenberg's uncertainty principle see *http://zebu.uoregon.edu/ imamura/122/feb9/hup.html* or use the alta vista search engine to find at least twenty other locations.

3. For a discussion of Satisfiability algorithms see Jun Gu, Paul W. Purdom, John Franco, Ben Wah, *Satisfiability Algorithms*, Cambridge University Press (1998).

[4] A set of clauses is minimally unsatisfiable if it is unsatisfiable and removal of any clause from the set leaves a satisfiable set of clauses